Filière : Licence Génie Informatique
Prof : A. BENMAKHLOUF

Les Bases de Données Réparties (BDR)

Table des matières

I- Introduction

De nos jours les entreprises et les différents organismes ont besoin de plus en plus de gérer un volume important de données. L'utilisation des bases de données centralisées avec un système de gestion des BDD génère une lourdeur dans la réalisation et l'exécuter à distance des différentes transactions surtout lorsqu'il s'agit des entreprises multinationales.
Pour surmonter ce problème, le recours à des bases de données réparties devient une nécessité.

II- Objectifs & Définitions

II-1 Objectifs des bases de données réparties

Les bases de données réparties ont une architecture qui peut s'adapter à une décentralisation des ressources des entreprises. Les objectifs des BDR sont :

Une indépendance à la localisation : permet l'exploitation des données dans les BDD relationnelles indépendamment de la localisation des données. c.-à-d., exprimées des requêtes en SQL similaires aux requêtes locales et simplifier la vue utilisateur et l'écriture de requêtes.

Une fiabilité élevée : permet de minimiser le risque lié à la centralisation des données dans un seul serveur. Les bases de données réparties ont souvent des données répliquées. La panne d'un site n'est pas très importante pour l'utilisateur, qui s'adressera à autre site.

Des performances : réduire le trafic sur le réseau est une possibilité d'accroître les performances. Le but de la répartition des données est de les rapprocher de l'endroit où elles sont accédées. Répartir une base de données sur plusieurs sites permet de répartir la charge sur les processeurs et sur les entrées/ sorties.

II-2 SGBD Répartie

Une base de données centralisée est gérée par un seul SGBD, est stockée dans sa totalité dans un emplacement physique unique et ses divers traitements sont confiés à une seule et même unité de traitement. Par opposition, une base de données distribuée est gérée par plusieurs processeurs, sites et SGBD.

II-3 Avantages et Inconvenants

Les principaux avantages d'une BDD_R sont :

1. Transparence pour l'utilisateur
2. Autonomie de chaque site
3. Absence de site privilégié
4. Continuité de service
5. Transparence vis à vis de la localisation des données
6. Transparence vis à vis de la fragmentation
7. Transparence vis à vis de la réplication
8. Traitement des requêtes distribuées
9. Indépendance vis à vis du matériel
10. Indépendance vis à vis du système d'exploitation
11. Indépendance vis à vis du réseau
12. Indépendance vis à vis du SGBD

Les principaux inconvenants sont :

1. Coût : la distribution entraîne des coûts supplémentaires en termes de communication, et en gestion des communications (hardware et software à installer pour gérer les communications et la distribution).
2. Problème de concurrence.
3. Sécurité : la sécurité est un problème plus complexe dans le cas des bases de données réparties que dans le cas des bases de données centralisées.

III- Conception d'une base de données répartie

La création d'un schéma de répartition est la partie la plus délicate de la phase de conception d'une BDD_R car il n'existe pas de méthode standard pour trouver la solution optimale. L'administrateur doit donc prendre des décisions en fonction de critères techniques et organisationnels avec pour objectif de minimiser le nombre de transferts entre sites, les temps de transfert, le volume de données transférées, les temps moyens de traitement des requêtes, le nombre de copies de fragments, etc.... on peut par ailleurs distinguer deux méthodes de répartition :

III-1 Conception descendante (top down design)

Dans cette méthode on définit dans un premier temps un schéma conceptuel global de la base de données répartie, puis on distribue sur les différents sites les schémas conceptuels locaux.
La répartition se fait donc en deux étapes, en première étape la fragmentation, et en deuxième étape l'allocation de ces fragments aux sites. Cette méthode (top down) est utilisée dans le cas où on veut faire la conception d'une nouvelle base de données dont les données vont être réparties dans les différents sites.

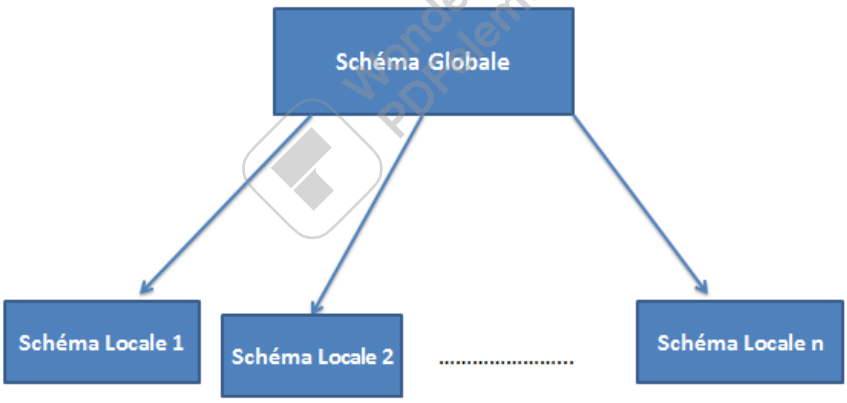


Fig-1 : Conception descendante

III-2 Conception ascendante (bottom up design)

Dans cette méthode il faut réussir à regroupe les schémas conceptuels déjà existant locaux dans un schéma conceptuel global. Cette approche est utilisée si on a déjà des données qui sont réparties dans des bases de données existantes et on veut les intégrées dans une base de données centrale et globale.

L'utilisation de cette approche est moins fréquente puis qu'elle difficile à mettre en œuvre. En effet, les BDDs existante sont en générale hétérogènes et sont gérés par déférents SGBD. Donc nécessité d'uniformiser puis de consolider les données des différents sites. Pour faire on doit par exemple :

- Identifier les données identiques
- Accorder leurs types
- Gérer leur cohérence…
- Interfacer ou adapter les SGBD…

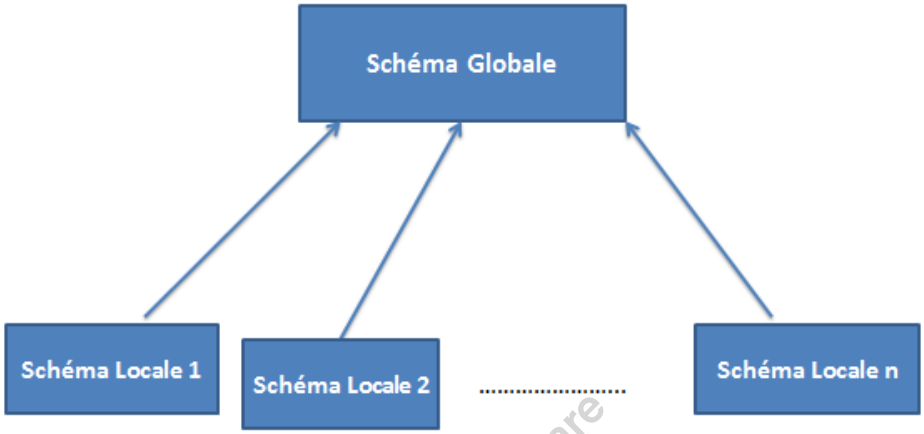


Fig-2 : Conception ascendante

III-3 Base de données fédérées & système multi base

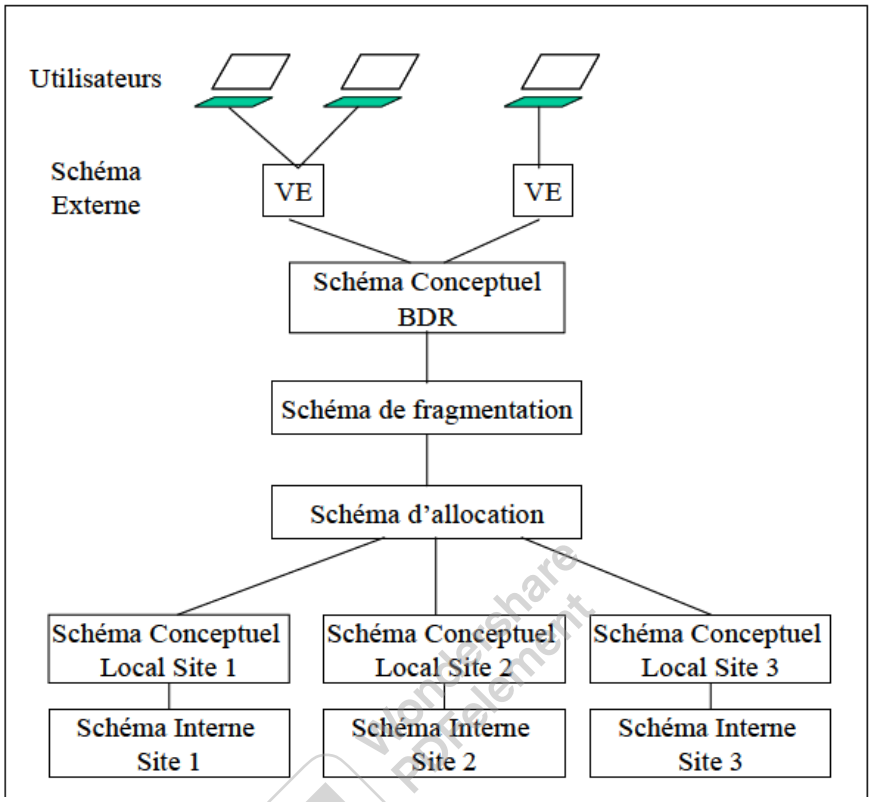
Les Base de données fédérées & les systèmes multi base sont différents des BDD réparties. En effet :

Dans une **BDD fédérée** chaque site a son schéma local. Ces schémas ne sont pas forcément tous inclus dans le schéma global. Ce dernier peut être obtenu par *une* approche ascendante partielle. Il y a donc un site central qui gère les données qui sont concernées par le schéma global.
Dans un **système multi-bases** il n'y a pas de schéma global, pas de site central. L'accès à une partie ou à la totalité des données peut de faire à distance via une application qui utilise un langage commun.

III-4 Architecture d'une BDR

En plus de la répartition physique des données, la répartition d'une base de données intervient dans les trois niveaux de son architecture.

Niveau externe : les vues sont distribuées sur les sites utilisateurs.
Niveau conceptuel : le schéma conceptuel des données est associé, par l'intermédiaire du schéma de répartition (lui-même décomposé en un schéma de fragmentation et un schéma d'allocation), aux schémas locaux qui sont réparties sur plusieurs sites, les sites physiques.
Niveau interne : le schéma interne global n'a pas d'existence réelle mais fait place à des schémas internes locaux répartis sur différents sites.



IV- Fragmentation

La fragmentation est le processus de décomposition d'une base de données en un ensemble de sous-bases de données. Cette décomposition doit être faite sans perte de données ou/et d'information. Il y a deux règles principales pour assurer une fragmentation sans cette perte de données.

La complétude : pour toute donnée d'une relation R, il existe un fragment R_i de la relation R qui possède cette donnée.

La reconstruction : pour toute relation décomposée en un ensemble de fragments Ri, il existe une opération de reconstruction.

IV-1 Types de Fragmentation

Il existe plusieurs techniques de fragmentation :

IV-1-1- fragmentation par objet

Cette technique consiste en la répartition des objets dans les différents serveur objet par objet. Toutes l'occurrences d'une classe (n-uplet d'une relation) appartient ainsi au même fragment.

- L'opération de fragmentation sera donc la définition des sous-schémas

- L'opération de recomposition des informations sera l'union entre les objets des sous-schémas à l'aide des jointures.

Exemple : pour le schéma de BDD suivant :

Client(NoClient, NomClient, PrénomClient, VilleClient, Age)
Agence(NomAgence, Adresse, Ville)
Compte(NCompte, TypeCompte, Solde, NoClient, NomAgence)

Ce schéma peut être fragmenté de la manière suivante :

$\{Clients,\ Comptes\}$ dans le serveur-1
$\{Agences\}$ dans le serveur-2

IV-1-2- fragmentation par occurrences (fragmentation horizontale FH)

Les tuples d'une même classe vont être réparties sur plusieurs Serveurs.

- L'opération de fragmentation sera donc la sélection (σ)
- L'opération de recomposition des informations sera l'union des sous-classes (U)

Exemple : dans l'exemple précédent on peut réaliser la FH suivante :

$$Agence1 = \sigma_{Ville='Casablanca'}(Agence)$$

$$Agence2 = \sigma_{Ville='\ Rabat\ '}(Agence)$$

IV-1-3- fragmentation par attributs (fragmentation verticale FV)

Les attributs de la même classe sont fragmentés dans plusieurs serveurs. Dans ce cas toutes les valeurs des occurrences pour un même attribut se trouvent dans le même fragment. Des attributs sont dupliqués dans les fragments pour assurer leurs jointures.

- L'opération de fragmentation sera donc la projection (π)
- L'opération de recomposition des informations sera les jointures ou les semi-jointure entre sous-classes ($\bowtie$) ou ($\ltimes$)

Exemple : dans l'exemple précédent on peut réaliser la FV suivante :

$$Client1 = \pi_{[NoClient,\ NomClient,\ PrenomClient]}(Client)$$
$$Client2 = \pi_{[NoClient,\ VilleClient,\ AGE]}(Client)$$

La recomposition de la table Client est réalisée par l'opération : $Client1 \bowtie Client2$

IV-1-4- fragmentation hybride (FHY)

Dans cette fragmentation seulement quelques valeurs des occurrences pour un même attribut se trouvent dans le même fragment. C'est donc une combinaison entre la FH et la FV.

- L'opération de partitionnement est une combinaison de projections et de sélections.
- L'opération de recomposition est une combinaison de jointures et d'unions.

Exemple : dans l'exemple précédent on peut réaliser la FHY suivante :

$$Client1 = \pi_{[NoClient,\ NomClient,\ PrenomClient]}(\sigma_{AGE\leq 40}(Client))$$
$$Client2 = \pi_{[NoClient,\ VilleClient,\ AGE]}(\sigma_{AGE\leq 40}(Client))$$
$$Client3 = \pi_{[NoClient,\ VilleClient,\ AGE]}(\sigma_{AGE>40}(Client))$$
$$Client4 = \pi_{[NoClient,\ NomClient,\ PrenomClient]}(\sigma_{AG>40}(Client))$$

La recomposition de la table Client est réalisée par l'opération :

$$(Client1 \bowtie Client2) \cup (Client3 \bowtie Client4)$$

IV-1-5- fragmentation des occurrences connexes (fragmentation dérivée FD)

La fragmentation d'une relation R s'accompagne avec une fragmentation des relations jointes Rj. Cette dernière se fait en générale en utilisant des semi jointures $R \ltimes R_j$.

Exemple :
Fragmenter dans un autre compte utilisateur les tables clients, comptes et Agences de la base de données $\{Clients, Comptes, Agences\}$ selon la sélection suivante :

$$Clients1: \sigma_{villeClient='Casablanca'}(Clients)$$

Réponses :

```
create table clients as (select * from BANK.clients where
villeclient='Casablanca');

create table comptes as (select * from BANK.comptes
where idclient in (select idclient from clients));
create table agences as (select * from BANK.agences where
idagence in
```

```
(select idagence from comptes));
```

Les contraintes d'intégrités

```
alter table clients add constraint c1 PRIMARY KEY (idclient);
alter table comptes add constraint c2 PRIMARY KEY (idcompte);
alter table agences add constraint c3 PRIMARY KEY (idagence);

alter table comptes add constraint f1 FOREIGN KEY (idclient)
REFERENCES clients(idclient);
alter table comptes add constraint f2 FOREIGN KEY (idagence)
REFERENCES agences(idagence);
```

Exercice2 :

Proposer un schéma de fragmentation horizontale des tables « clients » et « comptes » en tenant compte des requêtes suivantes :

$$R_1 = \sigma_{VilleClient='Casablanca' \ AND \ AGE<40}(Clients)$$
$$R_2 = \sigma_{VilleClient='Casablanca' \ AND \ Solde<0}(Clients \bowtie Comptes)$$
$$R_3 = \sigma_{VilleClient='Rabat'}(Clients)$$

Solution : le schéma sera réparti de la manière suivante :

Soient les variables logiques suivantes :

$$a = VilleClient =' Casablanca'$$
$$a' = VilleClient =' Rabat'$$
$$b = AGE < 40$$
$$c = Solde < 0$$

$$\boldsymbol{BDD1}: \begin{cases} Clients1 = \sigma_{a.b}(Clients) \\ Comptes1 = (Clients1 \ltimes Comptes) \end{cases}$$

$$\boldsymbol{BDD2}: \begin{cases} Comptes2 = \sigma_{a.c}(Clients \bowtie Comptes) \\ Client2 = (Clients \ltimes Comptes2) \end{cases}$$

$$\boldsymbol{BDD3}: \begin{cases} Clients3 = \sigma_{a'}(Clients) \\ Comptes3 = (Clients3 \ltimes Comptes) \end{cases}$$

IV-2-3 Définition des fragments verticaux d'une classe

Les fragments verticaux sont exclusifs, sauf en ce qui concerne le (ou les) attribut(s) de jointure qui sont insérés dans tous les fragments, et ce pour une décomposition sans perte d'information.

On procède comme suit,

1. Extraire toutes les expressions de projection concernées par les requêtes.
2. Ajouter à chacune d'elles le(s) attribut(s) de jointure si elle ne les possède pas.
3. Pour ne pas perdre des données, on Génère le complément de chaque expression (c'est à dire les autres attributs) en ajoutant le (ou les) attribut(s) de jointure.
4. Pour chaque fragment **F**i produit par la fragmentation horizontale, on recherche toutes les requêtes qui concernent ce fragment et à prendre les expressions de projection de ces requêtes.

Exercice :

Exercice : Soit la base de données relationnelles suivantes :

Client(NoClient, NomClient, PrénomClient, VilleCLient, Age)
Agence(CodeAgence, NomAgence, Adresse, Ville)
Compte(IdCompte, NCompte, Solde, NoClient, CodeAgence)

Proposer un schéma de fragmentation horizontale en tenant compte des requêtes suivantes :

$$R_1 = \pi_{[NoClient,NomClient,PrénomCLient]}(\sigma_{VilleClient='Casablanca' \text{ AND } Solde<0}(Clients \bowtie Comptes))$$

$$R_2 = \pi_{[NoClient,VilleClient,AGE]}(\sigma_{VilleClient='Casablanca' \text{ AND } Solde\geq=0}(Clients \bowtie Comptes))$$

La fragmentation est :

$$\boldsymbol{BDD1}: \begin{cases} Comptes1 = \sigma_{VilleClient='Casablanca' AND\ Solde<0}(Clients \bowtie Comptes) \\ Client1 = \pi_{[NoClient,NomClient,PrénomCLient]}(Clients \ltimes Comptes1) \end{cases}$$

$$\boldsymbol{BDD2}: \begin{cases} Comptes2 = \sigma_{VilleClient='Casablanca' AND\ Solde\geq0}(Clients \bowtie Comptes) \\ Client2 = \pi_{[NoClient,VilleClient,AGE]}(Clients \ltimes Comptes2) \end{cases}$$

$$\boldsymbol{BDD3}: \begin{cases} Clients3 = \sigma_{VilleClient!='Casablanca'}(Clients) \\ Comptes3 = (Comptes \ltimes Clients3) \end{cases}$$

IV- 3 Processus d'allocation des fragments.

L'affectation des fragments sur les sites est décidée en fonction de l'origine prévue des requêtes qui ont servi à la fragmentation. Le but est de placer les fragments pour minimiser les transferts de données entre les sites.

IV-3-1- Types d'allocation

a- Allocation sans réplication

Un fragment ne peut exister que dans un seul site. Ici, chaque donnée est mise à jour sur un seul site. Cette méthode est plus efficace quand les données sont beaucoup plus modifiées que lues.

b- Allocation avec réplication des données

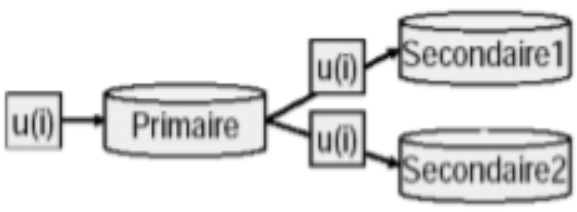
Pour des raisons de fiabilité et d'optimisation on peut décider de répliquer les données sur tous les sites. Ainsi, si un site est temporairement ou définitivement défaillant, on utilise simplement un autre. Il existe deux types de réplication

a. Types de réplication

Dans le cas d'une allocation avec duplication, La diffusion des mises à jour appliquée à une copie aux autres copies doit être assurée automatiquement par le SGBD réparti. Cette diffusion peut être basées sur la diffusion de l'opération de mise à jour ou sur la diffusion du résultat de l'opération. Un ordonnancement identique des mises à jour dans tous les sites est nécessaire afin d'éviter les pertes de mises à jour.

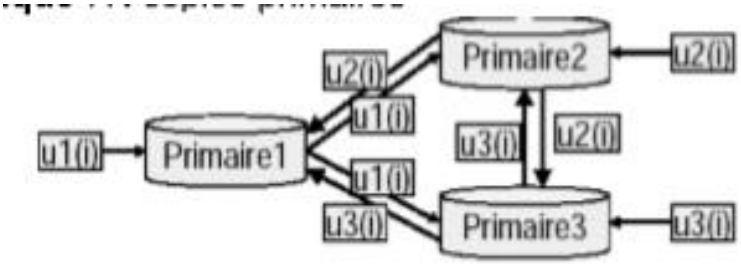
i. Réplication asymétrique

Un site primaire seul autorisé à mettre à jour et chargé de diffuser les mises à jour aux autres copies dites secondaire. Le site primaire effectue les contrôles et garantit l'ordonnancement correct des mises à jour. On distingue l'asymétrique synchrone et l'asymétrique asynchrone.



ii. *Réplication symétrique*

A l'opposé de la réplication asymétrique on ne privilégie aucune copie. Chaque copie peut faire la mise à jour à tous instant et assure la diffusion des mises à jour aux autres copies. On distingue aussi la symétrique synchrone et la symétrique asynchrone.



iii. *Réplication synchrone (synchronous replication) :*

Dans ce cas Une transaction est validée seulement lorsque tous les sites ont été mis à jour. Ce type de réplication peut se faire à l'aide **des triggers**.

Il est donc nécessaire de concevoir des algorithmes permettant aux triggers d'aiguiller les écritures faites dans une base de données vers les fragments concernés. Ces algorithmes doivent cerner toutes les conditions utilisées dans la fragmentation et dans la duplication toute en gardant la cohérence et l'intégrité de l'information.

Exercice : Soient les fragments de la BDD Bank :

$$\boldsymbol{BDD1}: \begin{cases} Comptes1 = \sigma_{VilleClient='Casablanca' \, AND \, Solde<0}(Clients \bowtie Comptes) \\ Client1 = (Clients \ltimes Comptes1) \end{cases}$$

$$\boldsymbol{BDD2}: \begin{cases} Comptes2 = \sigma_{VilleClient='Rabat' \, AND \, Solde\geq 0}(Clients \bowtie Comptes) \\ Client2 = (Clients \ltimes Comptes2) \end{cases}$$

La répartition avec réplication des fragments sera faite dans trois sites distants suivant le schéma de la figure-2. Pour assurer des opérations d'écriture synchronisées dans les différents fragments on doit créer deux types de triggers.

- Des *triggers de duplication* qui permettent de synchroniser les insertions les suppressions et les mises à jour du SiteB1.BDD1vers les fragments répliqués SiteB2.BDD1.
- Des *triggers de répartitions* qui permettent d'automatiser les écritures faites dans la BDD globale vers les fragments concernés.

En utilisant une répartition asymétrique, une proposition de répartition des différents triggers dans les trois sites est donnée par la figure-3.

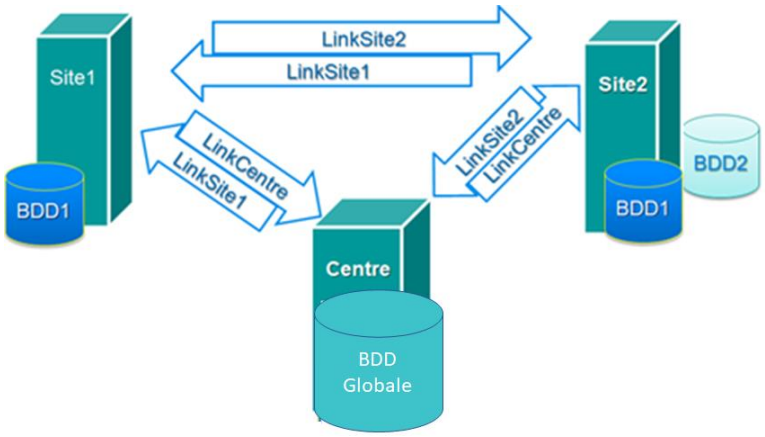


Figure-2 : Schéma de répartition de la base BDD

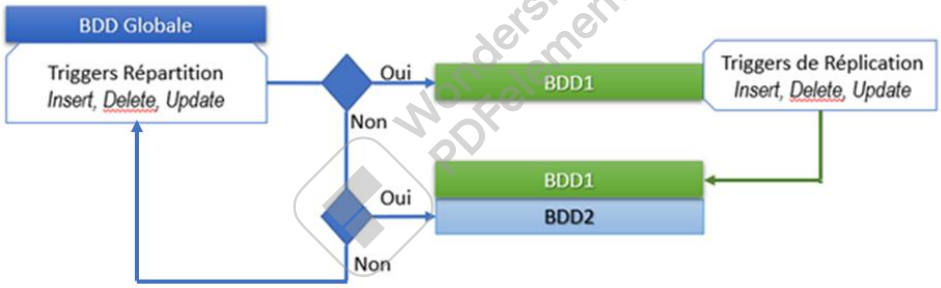


Figure-3 : Schéma de répartition des triggers dans les différents Sites dans le cas d'une répartition asymétrique avec réplication des fragments

Fragmentation de la BDD

Dans le compte utilisateur bank1on implémente le fragment BDD1

```
create table comptes1 as (select comptes.* from BANK.clients,
BANK.comptes
where clients.idclient=comptes.idclient
and villeclient='Casablanca' and solde<0);
create table clients1 as (select * from BANK.clients where
idclient in
(select idclient from comptes1));

alter table clients1 add constraint c1 PRIMARY KEY (idclient);
alter table comptes1 add constraint c2 PRIMARY KEY (idcompte);
alter table comptes1 add constraint f1 FOREIGN KEY (idclient)
REFERENCES clients1(idclient);
```

Dans le compte utilisateur bank2 on implémente le fragment BDD2

```
create table comptes2 as (select comptes.* from BANK.clients,
BANK.comptes
where clients.idclient=comptes.idclient
and villeclient='Rabat' and solde>=0);
create table clients2 as (select * from BANK.clients where
idclient in
(select idclient from comptes2));

alter table clients2 add constraint c1 PRIMARY KEY (idclient);
alter table comptes2 add constraint c2 PRIMARY KEY (idcompte);
alter table comptes2 add constraint f1 FOREIGN KEY (idclient)
REFERENCES clients2(idclient);
```

Dans le même compte utilisateur bank2 on duplique le fragment BDD1

```
create table clients1 as (select * from bank1.clients1);
create table comptes1 as (select * from bank1.comptes1);

alter table clients1 add constraint a1 PRIMARY KEY (idclient);
alter table comptes1 add constraint a2 PRIMARY KEY (idcompte);
alter table comptes1 add constraint g1 FOREIGN KEY (idclient)
REFERENCES clients1(idclient);
```

Dans bank1, on crée Les triggers de la synchronisation de la duplication de BDD1 dans BDD2:

```
create or replace TRIGGER Synchron_replication_Clients
BEFORE INSERT OR DELETE OR UPDATE on clients1
for EACH ROW
BEGIN
    DECLARE
    BEGIN
if INSERTING then
    INSERT INTO bank2.clients1 VALUES
(:NEW.IDclient,:NEW.Nomclient,:NEW.prenomclient,:NEW.villeCLIE
NT, :NEW.age);
elsif DELETING then
   DELETE bank2.clients1 WHERE IDclient=:OLD.IDclient;
elsif updating then
   UPDATE bank2.clients1 set Nomclient=:NEW.Nomclient,
   prenomclient=:NEW.prenomclient,
   villeCLIENT=:NEW.villeCLIENT,
   age=:NEW.age
```

```
   where idclient=:new.idclient;
end if;
    END;
END;

create or replace TRIGGER Synchron_replication_Comptes
BEFORE INSERT OR DELETE OR UPDATE on Comptes1
for EACH ROW
BEGIN
    DECLARE
    BEGIN
if INSERTING then
    INSERT INTO bank2.comptes1 VALUES
(:NEW.IDCOMPTE,:NEW.NUMCOMPTE,:NEW.SOLDE,:NEW.IDCLIENT,
:NEW.IDAGENCE);
elsif DELETING then
   DELETE bank2.comptes1 WHERE IDCOMPTE=:OLD.IDCOMPTE;
elsif updating then
   UPDATE bank2.comptes1 set Solde=:New.Solde where
idcompte=:new.idcompte;
end if;
    END;END;
```

Le trigger de la répartition de l'insertion d'un nouveau compte dans BDD1 et BDD2 est donné par :

```
create or replace trigger syn_insert_comptes
BEFORE INSERT ON comptes
FOR EACH ROW
begin
declare
V clients.villeclient%type;
S comptes.solde%type:=:new.solde;
n integer;
R clients%rowtype;
begin
select villeclient into V
from clients where idclient=:new.idclient;
if(V='Casablanca') then
    select count(idclient) into n from bank1.clients1
    where idclient=:new.idclient;
    if(n=0) then
        select * into R from clients where
        idclient=:new.idclient;
        insert into BANK1.clients1 values R;
    end if;
```

```
    insert into bank1.comptes1 values
    (:new.idcompte, :new.numcompte, :new.solde,
    :new.idclient, :new.idagence);
ELSif (V='Rabat' and :new.solde>=0) then
    select count(idclient) into n from bank2.clients2
    where idclient=:new.idclient;
    if(n=0) then
        select * into R from clients where
        idclient=:new.idclient;
        insert into BANK2.clients2 values R;
    end if;
    insert into bank2.comptes2 values
    (:new.idcompte, :new.numcompte, :new.solde,
    :new.idclient, :new.idagence);
end if;
end;
end;
```

iv. Réplication asynchrone (asynchronous replication) :

Les sites de réplication sont mis-à-jour en différé, à partir de la copie primaire, après la confirmation de la transaction. Ce type de réplication peut se faire à l'aide des ***vues matérialisées.***

syntaxe :

CREATE MATERIALIZED VIEW TableCopie [REFRESH {FAST|COMPLETE|FORCE} [ON COMMIT|ON DEMAND]] [FOR UPDATE] AS Select * from Table

V- Les Vues Matérialisées (Materialized Views).

Les vues matérialisée (VM) permettent de créer des vues physiques d'une table ou d'une requête SQL sur un ensemble de table. La différence d'une vue standard c'est que les données sont dupliquées. On l'utilise pour, des fins d'optimisation de performance, lorsqu'une requête demande l'exécution d'un ensemble de sous-requêtes, ou pour faire des réplications de table.
La 'fraicheur' des données de la VM dépend des options choisies. Le décalage entre les données de la table maître et la VM peut être nul (rafraichissement synchrone) ou d'une durée planifiée : heure, jour,etc. Les vues matérialisées peuvent être utilisées directement par l'optimiseur, afin de modifier les chemins d'exécution des requêtes. Pour ce, il faut disposer du privilège QUERY REWRITE pour pouvoir réécrire la requête, et que QUERY_REWRITE_ENABLED soit TRUE
Syntaxe:
(ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE).

Les VMs restent, par conséquent, un moyen très simple pour répliquer et dupliquer les données d'une base de données dans d'autres base de données stockées dans des sites distants. Dans ce cas les mises à jour des opérations d'écriture seront effectuées en différé c-a-d les écritures seront validées dans la base de données où on a réalisé ces écritures avant d'être validées dans les sites distants.

V-1 Les types de vues matérialisées

Il existe deux types de VM :

- Les VMs sur clé primaire. C'est le type **par défaut**

```
CREATE MATERIALIZED VIEW NOM_VM
REFRESH with primary key
AS SELECT * FROM ….
```

- Les VMs sur rowid. Utile lorsque la vue ne contient pas de (ou pas toutes les colonnes de la) clé primaire

```
CREATE MATERIALIZED VIEW NOM_VM
REFRESH with rowid
AS SELECT * FROM ….
```

Contraintes (assez fortes !). Ce type de VM :

- Ne permet pas le fast refresh s'il n'a pas eu un complet avant.
- Est interdit si on a SELECT avec distinct, group by, connect by, fonctions d'aggregats, opérateurs ensemblistes, sous requêtes.

V-2 Les méthodes de rafraichissement de la VM

Dans cette partie nous expliquons comment actualiser les vues matérialisées, élément clé pour maintenir de bonnes performances et des données cohérentes lors de l'utilisation de vues matérialisées dans un environnement d'entrepôt de données ou dans le cas des bases de données réparties (donnée dupliquées). Dès que certaines données de la BDD ont été mise à jour. Toutes les vues matérialisées utilisant ces données deviennent alors obsolètes. L'opération de rafraîchissement des VMs doit être lancée pour actualiser les données des VMs. Il existe deux méthodes pour rafraichir : *Fast* et *Complet*

FAST : mise-à-jour partielle de la copie locale. C'est la méthode la plus efficace car au lieu d'avoir à recalculer l'intégralité de la vue matérialisée, les modifications sont appliquées aux données concernées par les modifications. Les VMs utilisent des journaux spécifiques traçant les modifications de la table maître. Lorsque des changements sont effectués sur les données des tables maîtres, Oracle stocke les lignes qui dérivent ces changements dans le journal de vue matérialisée (fichiers Log), et utilise ensuite ces changements dans le journal de vue matérialisé pour réactualiser les vues matérialisées par rapport à la table maître. Un journal de vues matérialisées est un objet de schéma qui enregistre les modifications apportées à une table de base afin qu'une vue matérialisée définie sur la table de base puisse être actualisée de

manière incrémentielle. *Chaque journal de vues matérialisées est associé à une seule table de base*. Le journal des vues matérialisées réside dans la même base de données et le même schéma que sa table de base.

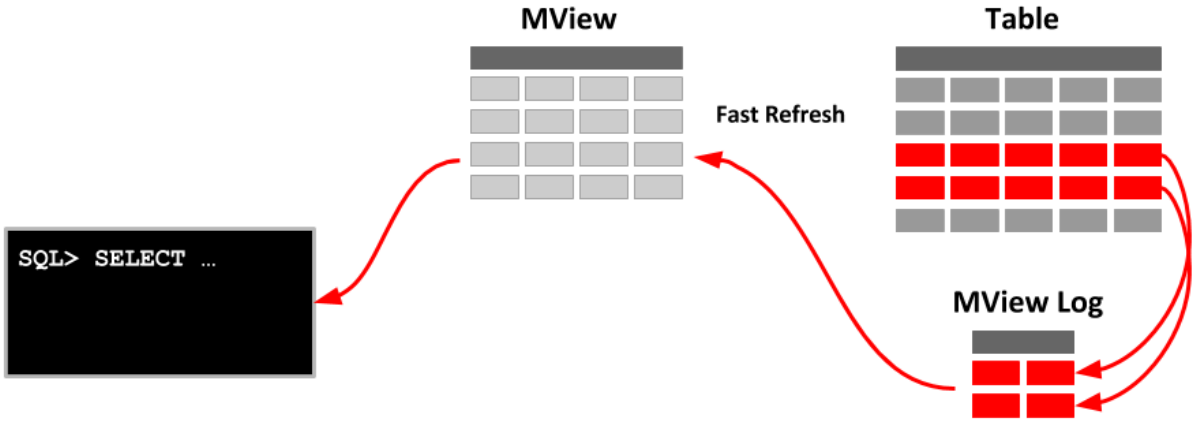


Figure-4 : Rafraichissement d'une VM en utilisant une journalisation stockée dans un fichier Log.

Syntaxe de création d'une journalisation sur une table :

```
create Materialized View Log on Table_name
with Rowid (liste attributs), Primary key including new
values;
```

with Rowid, Primary key: inclure la colonne de clé primaire et le rowids de toutes les lignes modifiées de la table de base dans un journal de vues matérialisées

Liste Attributs : la liste des attributs de la table qui sont utilisés par la requête de la VM

Including new values : permet à la BDD d'enregistrer, dans le journal des VMs, à la fois les anciennes et les nouvelles valeurs pour les opérations de mise à jour LMD. Concerne une table sur laquelle vous disposez d'une vue agrégée matérialisée à table unique

Conditions pour appliquer un rafraîchissement rapide sur VM
Ci-dessous quelques restrictions que doit vérifier la requête de définition de la VM :

- La vue matérialisée ne doit pas contenir de références à des expressions non répétitives telles que SYSDATE et ROWNUM.
- Elle ne peut pas contenir de SELECT sous-requête de liste.
- Elle ne peut pas contenir de fonctions analytiques (par exemple, RANK) dans la SELECT clause.
- Elle ne peut pas contenir de HAVING clause avec une sous-requête.
- Elle ne peut pas contenir des requêtes imbriquées qui ont ANY, ALL ou NOT EXISTS.
- Elle ne peut pas contenir de [START WITH …] CONNECT BY clause.
- Elle ne peut pas contenir plusieurs tableaux détaillés sur différents sites.
- La vue matérialisée à la validation ne peut pas avoir de tables de détails distantes.

- Les vues matérialisées imbriquées doivent avoir une jointure ou un agrégat.

Conditions pour appliquer un rafraîchissement rapide sur les VMs avec jointures uniquement

La définition de requêtes pour des vues matérialisées avec des jointures uniquement et sans agrégats a les restrictions suivantes sur l'actualisation rapide :

- Toutes les restrictions générales sur le rafraîchissement rapide.
- Elles ne peuvent pas avoir de GROUP BY clauses ou d'agrégats.
- Les Rowids de toutes les tables de la FROM liste doivent apparaître dans la SELECT liste de la requête.
- Les journaux de vues matérialisées doivent exister avec des rowids pour toutes les tables de base de la FROM liste de la requête.

Conditions pour appliquer un rafraîchissement rapide sur les VMs avec agrégation

La définition de requêtes pour des vues matérialisées avec des agrégats ou des jointures a les restrictions suivantes sur l'actualisation rapide :

- Toutes les restrictions générales sur le rafraîchissement rapide.
- Toutes les tables de la vue matérialisée doivent avoir des journaux de vue matérialisée, et ces journaux doivent :
    - Contenir toutes les colonnes de la table référencée dans la vue matérialisée.
    - Spécifier avec `ROWID` et `INCLUDING NEW VALUES`.
    - Spécifier la `SEQUENCE` clause si la table doit contenir un mélange d'insertions/chargements directs, de suppressions et de mises à jour.
- Seuls `SUM`, `COUNT`, `AVG`, `STDDEV`, `VARIANCE`, `MIN` et `MAX` sont pris en charge pour une actualisation rapide.
- `COUNT(*)` doit être spécifié.
- Pour chaque agrégat tel que `AVG(expr)`, `SUM(expr)`, le correspondant `COUNT(expr)` doit être présent.
- Si `VARIANCE(expr)` ou `STDDEV(expr)` est spécifié `COUNT(expr)` et `SUM(expr)` doit être spécifié.
- La `SELECT` colonne de la requête de définition ne peut pas être une expression complexe avec des colonnes de plusieurs tables de base. Une solution de contournement possible consiste à utiliser une vue matérialisée imbriquée.
- La `SELECT` liste doit contenir toutes les `GROUP BY` colonnes.
- Si la vue matérialisée présente l'un des éléments suivants, l'actualisation rapide n'est prise en charge que sur les insertions.
    - Vues matérialisées avec `MIN` ou `MAX` agrégats
    - Vues matérialisées qui n'ont `SUM(expr)` pas `COUNT(expr)`
    - Vues matérialisées sans `COUNT(*)`

  Une telle vue matérialisée est appelée vue matérialisée à insertion uniquement.
- Une vue matérialisée avec `MAX` ou `MIN` est rapidement actualisable après suppression ou instructions DML (Data Manipulation Langage) mixtes si elle n'a pas de `WHERE` clause.
- S'il n'y a pas de jointures externes, vous pouvez avoir des sélections et des jointures arbitraires dans la `WHERE` clause.

- Les vues agrégées matérialisées avec jointures externes peuvent être actualisées rapidement après des charges DML conventionnelles et directes, à condition que seule la table externe ait été modifiée. En outre, des contraintes uniques doivent exister sur les colonnes de jointure de la table de jointure interne. S'il existe des jointures externes, toutes les jointures doivent être connectées par `AND` et doivent utiliser l'opérateur d' égalité (=).
- Pour les vues matérialisées avec `CUBE`, `ROLLUP`, groupes de regroupement ou concaténation de celles-ci, les restrictions suivantes s'appliquent :
    - La `SELECT` liste doit contenir un identificateur de regroupement qui peut être soit une `GROUPING_ID`fonction sur toutes les `GROUP BY`expressions, soit `GROUPING`une fonction pour chaque `GROUP BY`expression. Par exemple, si la `GROUP BY`clause de la vue matérialisée est " `GROUP BY CUBE(a, b)`", la `SELECT`liste doit contenir soit " `GROUPING_ID(a, b)`" soit " `GROUPING(a) AND GROUPING(b)`" pour que la vue matérialisée puisse être actualisée rapidement.
    - `GROUP BY` ne doit pas donner lieu à des regroupements en double. Par exemple, " `GROUP BY a, ROLLUP(a, b)`" n'est pas actualisable rapidement car il en résulte des regroupements en double " `(a), (a, b), AND (a)`"

Exemple :

```
create Materialized View Log on commandes with Rowid, Primary
key (idclient) including new values;


Create Materialized View V1
Refresh Fast on Commit as
select idclient, count(idcommande) as nb
from commandes
group by Idclient;
```

with Rowid, Primary key : inclure la colonne de clé primaire et le rowids de toutes les lignes modifiées de la table de base dans un journal de vues matérialisées

Exemple -2:
Dans la BDD ci-dessous on veut dupliquer les données renvoyées par la requête qui calcule le total des soldes par client :

Client(NoClient, NomClient, PrénomClient, VilleClient, Age)
Agence(NomAgence, Adresse, Ville)
Compte(NCompte, TypeCompte, Solde, NoClient, NomAgence)

```
create Materialized View Log on comptes
with  primary key, Rowid (solde, idclient) including new
values;
CREATE MATERIALIZED VIEW LOG ON CLIENTS
WITH PRIMARY KEY, ROWID(NOMCLIENT)
```

```
INCLUDING NEW VALUES;

create materialized view SoldeTot_Client
refresh fast on commit as
select clients.idclient, sum(solde), count(*) as tot
from clients, comptes
where clients.idclient=comptes.idclient
group by clients.idclient;
```

Exemple-3
Dans la BDD gestion des ventes

clients(idclient, nomclient, afresse, villeclient)
commandes(idcommande, datecommande, idclient)
lignecommandes(idlignecommande, idcommande, idproduit, quantite, remise)
Produits(idproduit, Designation, prixunitaire, TVA)

On veut dupliquer les données renvoyées par la requête qui calcule le montant des ventes par clients, par produits et par datecommande:

```
create materialized view log on commandes
with primary key, rowid(idclient, Datecommande) including new
values;

create materialized view log on lignecommandes
with primary key, rowid(idcommande, quantite, remise,
idproduit)
including new values;

create materialized view log on produits
with primary key, rowid(prixunitaire) including new values;

create materialized view Table_Ventes
refresh Fast on commit as
select idclient, produits.idproduit, datecommande,
sum(prixunitaire*quantite*(1-remise/100)) as CA
from commandes, lignecommandes, produits
where commandes.idcommande=lignecommandes.idcommande
and lignecommandes.idproduit=produits.idproduit
group by idclient, produits.idproduit, datecommande;
```

COMPLETE : effectue le refresh complet en exécutant le SELECT de définition de la MV. Un rafraîchissement complet peut être demandé à tout moment au cours de la vie de toute vue matérialisée. L'actualisation implique la relecture des tableaux détaillés pour recalculer les résultats de la vue matérialisée. Cela peut être un processus très long, surtout s'il y a

d'énormes quantités de données à lire et à traiter. Par conséquent, on doit toujours tenir compte du temps nécessaire pour traiter une actualisation complète avant de la demander.

FORCE : **Cette méthode** effectue dans un premier temps une actualisation rapide (FAST). Si cette dernière échoue, une actualisation complète se produira.

Vérification de la méthode de rafraîchissement sur une VM :
Oracle dispose de la procédure dbms_Mview.explain_Mview('NomVM') qui permet de vérifier la méthode d'actualisation qui peut être appliquée sur une VM déjà créé. Cette procédure donne même les causes du non fonctionnement d'une méthode d'actualisation (Fast ou Complet).
Pour utiliser cette procédure il faut tout d'abord créer la table MV_CAPABILITIES

```
create table MV_CAPABILITIES_TABLE
(
  statement_id       varchar(30) ,
  mvowner            varchar(30) ,
  mvname             varchar(30) ,
  capability_name    varchar(30) ,
  possible           character(1) ,
  related_text       varchar(2000) ,
  related_num        number ,
  msgno              integer ,
  msgtxt             varchar(2000) ,
  seq                number
) ;
```

On exécute ensuite la procédure

```
begin
dbms_Mview.explain_Mview('NomVM');
end;
```

On peut afficher les données de la table MV_Capabilities_table :

```
SELECT capability_name, possible,
FROM mv_capabilities_table
WHERE capability_name like '%FAST%';
```

CAPABILITY_NAME	POSSIBLE
REFRESH FAST	N
REFRESH FAST AFTER INSERT	N
REFRESH FAST AFTER ONETAB DML	N
REFRESH FAST AFTER ANY DML	N
REFRESH FAST PCT	N

V-3 Les modes de rafraichissement de la VM

ON COMMIT: synchrone, rafraîchissement lorsqu'une transaction modifiant les tables maîtresses fait un commit. Ce mode de rafraichissement est utilisé que si ***la table maître est dans la même base de données où on a créé la vue matérialisée***. Par conséquent le mode « on commit » n'est pas pris en charge dans les bases de données distantes.

ON DEMAND *asynchrone (par défaut):* C'est un rafraîchissement sur demande de l'usager ou à un instant qui peut être spécifié avec les clauses START et NEXT. Ce mode est utilisé dans le cas des VM qui utilisent des tables maitresses distantes. L'actualisation peut être effectuée avec les méthodes d'actualisation fournies dans le DBMS_SYNC_REFRESH ou les DBMS_MVIEW packages :

V-4 Actualisation manuelle à l'aide du package DBMS_MVIEW

Lorsqu'une vue matérialisée est actualisée ON DEMAND, l'une des trois méthodes d'actualisation peut être spécifiée :

- Complete : présenté par le paramètre 'C'
- Fast : présenté par le paramètre 'F'
- Force : présenté par le paramètre ' ?'

a- Procédure `DBMS_MVIEW.REFRESH`

La Procédure DBMS_MVIEW.REFRESH est utilisée pour actualiser une ou plusieurs vues matérialisées. Les paramètres requis pour utiliser cette procédure sont :

```
DBMS_MVIEW.REFRESH('Liste des VMs', 'mode_Actu', '', Erreur, Atomique);
```

- **Liste des VMs :** La liste délimitée par des virgules des vues matérialisées à actualiser
- **Mod_Actu :** La méthode de rafraîchissement : '`F`'-Fast, '`?`'-Force, '`C`'-Complete
- **Erreur** (TRUE/FALSE): Actualiser après des erreurs : S'il est défini sur TRUE, le number_of_failuresparamètre de sortie est défini sur le nombre d'actualisations ayant échoué et un message d'erreur générique indique que des échecs se sont produits.
- **Atomique** (TRUE/FALSE): Rafraîchissement atomique : Si défini sur TRUE, toutes les actualisations sont effectuées en une seule transaction. Si la valeur est définie sur FALSE, chacune des vues matérialisées est actualisée de manière non atomique dans des transactions distinctes.

Exemple :

```
begin
DBMS_MVIEW.REFRESH('Table_Vente', 'F', 'True', 'False');
end;
```

V-5 Actualisation de toutes les vues matérialisées avec REFRESH_ALL_MVIEWS

Une alternative à la spécification des vues matérialisées à actualiser consiste à utiliser la procédure DBMS_MVIEW.REFRESH_ALL_MVIEWS(). Cette procédure actualise toutes les vues matérialisées. Si l'actualisation de l'une des vues matérialisées échoue, le nombre d'échecs est signalé.

```
Begin
DBMS_MVIEW.REFRESH_ALL_MVIEWS(pannes,'mod_actu','', Erreur, Atomique);
End;
```

- pannes : Le nombre d'échecs (il s'agit d'une OUT variable)
- mod_actu : La méthode de rafraîchissement : F-Fast, ?-Force, C-Complete
- '' : Le segment d'annulation à utiliser
- Actualiser après des erreurs (TRUE ou FALSE)
- Atomique : Rafraîchissement atomique (`TRUE`ou `FALSE`)

Exemple :

```
declare
X integer;
Begin
DBMS_MVIEW.REFRESH_ALL_MVIEWS(X,'C','', TRUE, FALSE);
end;
```

Exemple de fragmentation en utilisant les VMs :

Revenant à l'exemple précèdent de la BDD (clients, agences, comptes). La fragmentation peut se faire en utilisant les VMs. Ces derniers garantissent une répartition et une duplication asynchrone (différée) avec « on demand » vu que les requêtes des VMs utilisent des tables qui appartiennent à d'autre BDD (stockées dans d'autre serveurs).

Dans bank1 (BDD1) :

```
create materialized view clients1
refresh complete on demand as
select distinct clients.* from bankcenter.clients@liencenter,
bankcenter.comptes@liencenter
where clients.idclient=comptes.idclient
AND villeclient='Casablanca' AND solde<0;

create materialized view comptes1
refresh complete on demand as
select distinct comptes.* from bankcenter.clients@liencenter,
bankcenter.comptes@liencenter
where clients.idclient=comptes.idclient
```

```
AND villeclient='Casablanca' AND solde<0;
```

Pour demander le rafraîchissement :

```
execute dbms_mview.refresh('clients1');
execute dbms_mview.refresh('comptes1');
```

Dans bank2 (BDD2) :

```
create materialized view clients2
refresh complete on demand as
select distinct clients.* from bankcenter.clients@liencenter,
bankcenter.comptes@liencenter
where clients.idclient=comptes.idclient
AND villeclient='Rabat' AND solde>=0;

create materialized view comptes2
refresh complete on demand as
select distinct comptes.* from bankcenter.clients@liencenter,
bankcenter.comptes@liencenter
where clients.idclient=comptes.idclient
AND villeclient='Rabat' AND solde>=0;
```

Pour la duplication du fragment BDD1 dans le site-2

```
create materialized view clients1
refresh complete on demand as
select * from bank11.clients1@liensite1;

create materialized view comptes1
refresh complete on demand as
select * from bank11.comptes1@liensite1;
```

Pour forcer le rafraichissement

```
execute dbms_mview.refresh('clients2');
execute dbms_mview.refresh('comptes2');
execute dbms_mview.refresh('clients1');
execute dbms_mview.refresh('comptes1');
```

VI Les Requêtes Réparties

Les règles d'exécution et les méthodes d'optimisation de requêtes appliquées pour des BDD centralisé sont toujours valables, mais il faut prendre en compte, d'une part, la fragmentation et la répartition des données sur différents sites, et d'autre part le problème du coût des communications entre sites pour transférer les données.

Les requêtes d'insertion de mise à jour et de suppression vont être concernées par le problème de la fragmentation avec ou sans duplication tandis que les requêtes de sélection et de projection vont être concernées par le problème des coûts des communications entre les sites.

La complexité d'une requête dans une base de données répartie est définie en fonction des facteurs suivants :

- Entrées/ Sorties sur les disques, c'est le coût d'accès aux données.
- Coût CPU : c'est le coût des traitements de données pour exécuter les opérations algébriques (jointures, sélections ...).
- Communication sur le réseau : c'est le temps nécessaire pour échanger un volume de données entre des sites participant à l'exécution d'une requête.

Dans une base de données centralisée, seuls les facteurs E/Ss et CPU déterminent la complexité d'une requête.

VI-1 les requêtes d'Ecriture

Les requêtes d'écriture vont être concernées par le problème de la fragmentation avec ou sans duplication

- *Insertion*

Il faut retrouver le fragment horizontal concerné en utilisant les conditions qui définissent les fragments horizontaux, puis insertion du tuple dans tous les fragments verticaux correspondants.

- *Suppression*

Rechercher le tuple dans les fragments qui sont susceptibles de contenir le tuple concerné, et supprimer les valeurs d'attribut du tuple dans tous les fragments verticaux.

- *Modification*

Rechercher le tuple dans les fragments qui sont susceptibles de contenir le tuple concerné, et modifier les *valeurs* d'attribut du tuple dans tous les fragments verticaux. Les modifications peuvent être accompagnées par une suppression et/ou une insertion du tuple concerné dans les autres fragments.

Exemple : algorithme du trigger pour la réplication de la suppression des données dans les fragments répliqués :

```
create or replace TRIGGER NomTrigger
BEFORE DELETE on BDD1.Table
for EACH ROW
BEGIN
    DECLARE
    BEGIN
    DELETE from BDD2.Table WHERE
BDD2.Table.Attribut=:old.Attribut;
```

```
    DELETE from BDD3.Table WHERE
BDD3.Table.Attribut=:old.Attribut;
    ……
    DELETE from BDDn.Table WHERE
BDDn.Table.Attribut=:old.Attribut;
    END;
END;
```

Exemple2 : Algorithme du trigger de la réparation pour la mise à jour des données dans les fragments répartis :

Soient C1 et C2 les conditions de répartition des données dans respectivement dans « Site1 » et « Site2 ». Les données qui ne respectent pas C1 et C2 sont stockées dans « Site ».
L'algorithme du trigger responsable à la répartition de la mise à jour est donné par :

```
Si (OLD.données vérifient C1) alors
       Si(NEW.données vérifient C1) alors
            Update les données du Site1
       sinon
             DELETE les données du Site1
             Si(NEW.données vérifient C2) alors
                  INSERT les données dans Site2
             Fin SI
        Fin SI
Sinon Si (OLD.données vérifient C2) alors
       Si(NEW.données vérifient C2) alors
            Update les données du Site2
       sinon
             DELETE les données du Site2
             Si(NEW.données vérifient C1) alors
                  INSERT les données dans Site1
             Fin SI
        Fin SI
Sinon Si (NEW.données vérifient C1) alors
            INSERT les données dans Site1
       Sinon Si(NEW.données vérifient C2) alors
            INSERT les données dans Site2
Fin Si
```

VI-2 Les requêtes de Lecture

Dans le cas des requêtes locales le coût provient principalement au :

- Nombre Entrées/ Sorties sur les disques, c'est le coût d'accès aux données dans le disque.
- Coût CPU : c'est le coût des traitements de données pour exécuter les opérations algébriques (jointures, sélections ...).

À ces coûts s'ajoute celui de la Communication sur le réseau :

- C'est le temps nécessaire pour échanger un volume de données entre des sites participant à l'exécution d'une requête.

a- Optimisation des requêtes dans une BDD réparties.

Le parallélisme des opérations, c.-à-d. le calcule simultanément plusieurs opérations d'une même requête, reste avantageux dans le contexte réparti. Pour effectuer ces opérations il faut transférer entre les sites une quantité minimale de données.

Optimisation globale.

La règle est de transférer la plus petite table d'abord.

Soient des jointures entre les tables T1, T2 etT3. Chaque site « S_i » contient une seule table « T_i ».

Algo 1 :
Transférer T1 au site 2
R=T1 ⋈ T2 au site 2
Transférer R au site 3
R ⋈ T3 est le Résultat final au site 3

Algo 2 :
Transférer T2 au site 1
R=T1 ⋈ T2 au site 1
Transférer R au site 3
R ⋈ T3 est le Résultat final au site 3

Algo 3 :
Transférer T1 au site 3
Transférer T2 au site 3
T1 ⋈ T2 ⋈ T3 est le Résultat final au site 3

Si on suppose que le nombre de tuples de chaque relation est donné de telle sorte qu'on a : NT1<NT2<NT3. L'algorithme algo-3 sera donc le plus optimal.

Optimisation par semi-jointure

La réglé est d'effectuer une semi-jointure au lieu d'une jointure complète. le principe est d'effectuer deux petites jointures plutôt qu'une grosse; c'est à dire deux petites transmissions de données plutôt qu'une seule beaucoup plus volumineuse.

Pour effectuer la jointure T1⋈T2 :

Algo 1 (sans optimisation) :
Transférer T2 au site 1
T1 ⋈ T2 = Résultat final au site 1

Algo 2 (Optimisation par semi-jointure) :
Transférer $\pi_A(T1)$ au site 2
$T2 \bowtie \pi_A(T1) = T1 ⍰ T2 = R$
Transférer R au site 1
R ⋈ T1 = Résultat final au site 1

Il faut donc transférer au site2 seulement les colonnes de « T1 » nécessaires à la semi-jointure (ensemble A). Renvoyer ensuite au site1 le résultat de la semi-jointure pour compléter la jointure avec les autres colonnes.

Optimisation par Parallélisme inter-opération et inter-site

Soit les opérations : T1 ⋈ T2 ⋈ T3 ⋈ T4

Algorithme :
Transférer T2 au site1
R = T1 ⋈ T2 au site1
En parallèle, transférer T4 au site3
S = T3 ⋈ T4 au site 3
Transférer S au site 1
Ensuite, R ⋈ S = Résultat final au site 1

Exemple :

Soit le schéma d'une BDD

Clients(**codeclient**, Société, Adresse, Tel)
Commandes(**N°Commande**, DateCommande, codeClient)
Produits(**Réf**, Designation, Stock, PrixUnitaire)
DétailCommandes(**N°Commande, Réf,** Quantité)

Question traitée : Quels sont les clients de France qui ont commandé le produit « Tarte au sucre » avec une quantité par commande >20.

On suppose la répartition de la BDD suivantes :

- Table « clients » dans le Site1
- Table « commandes » dans le Site2
- Table « détailscommandes » dans le Site3

- Table « produits » dans le site4

Requête répartie non optimisé :

Transférer commandes au Site1 : R1 = clients ⋈ Commandes dans Site1
Transférer détailsCommandes au site1 : R2 = R1 ⋈ détailsCommandes dans Site1
Transférer produits au site1 : R3 = R2 ⋈ Produits.

```
select distinct clients.codeclient, société
FROM clients, commandes, détailscommandes, produits
where clients.codeclient=commandes.codeclient
AND commandes.ncommande=détailscommandes.ncommande
AND détailscommandes.refproduit=produits.refproduit
AND quantité>20 AND designation = 'Tarte au sucre'
AND pays='France';
```

Avec « Commandes » « détailsCommandes » et « Produits » sont des synonymes des chemins pour accéder aux tables distantes stockées dans les différents serveurs.

$Commandes \equiv userSite2.Commandes@lienSite2$
$DétailsCommandes \equiv userSite3.Commandes@lienSite3$
$Produits \equiv userSite4.Commandes@lienSite4$

Requête répartie avec optimisation par Parallélisme :

Transférer commandes au Site1 : G = clients ⋈ Commandes dans Site1
Transférer produits au site2 : D = DétailsCommandes ⋈ Produits dans Site2
Transférer D au site1 : R = G ⋈ D.

Dans le Site1 :

```
select clients.codeclient, société, ncommande
from clients inner join commandes
on clients.codeclient=commandes.codeclient
where pays='France'
```

Dans le Site 2 :

```
select détailscommandes.*
from détailscommandes inner join produits
on détailscommandes.refproduit=produits.refproduit
where quantité>20 AND designation='Tarte au sucre';
```

Dans le Site1 :

```
select distinct codeclient, société
from G inner join D on D.ncommande=G.ncommande;
```

VI- BD répartie avec Oracle

V-1 DATABASE LINKS

Un DBlink est un objet Oracle permettant de créer un lien entre plusieurs bases de données Oracle, ce lien peut être sur le même hôte, sur un hôte appartenant au même domaine ou sur tout autre hôte joignable par le protocole TCP/IP. Les BD link permettent de manipuler les d'une BDD répartie sur plusieurs sites distant comme si c'était une BDD centrale stocké dans un seul serveur. Pour créer un DBLink, il faut que les fichiers de configuration « tnsname.ora » et « lestener.ora » soit correctement renseigné sur le service de connexion à distance :

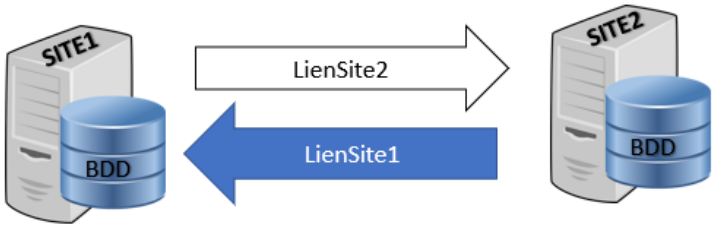
- Adresse : Protocol, Host et le Port
- Connexion des data : nom du serveur et le nom du service

Syntaxe :

```
CREATE [SHARED] [PUBLIC] DATABASE LINK nomLien CONNECT TO
compteUtilisateur IDENTIFIED BY "motDePasse" USING
'Host:serveur/nomService'
```

- SHARED: permet de partager la connexion entre plusieurs usagers
- PUBLIC : rend le lien disponible à tous les usagers locaux
- nomService : Spécifie le nom de service d'une base de données distante. Si on spécifie uniquement le nom de la base de données, Oracle ajoute implicitement le domaine de la base de données à la chaîne de connexion pour créer un nom de service complet. Par conséquent, si le domaine de la base de données distante est différent de celui de la base de données actuelle, on doit spécifier le nom du service complet. Le nonService doit être défini dans un fichier de configuration de la BDD Oracle (Listener.ora et tnsname.ora).

Exemple : soit deux sites distants appartements à un réseaux :



Dans Site1:

CREATE PUBLIC DATABASE LINKSITE2
CONNECT TO USERSITE2 IDENTIFIED BY "1111" USING 'SITE2';

On peut écrire la requête suivante pour interroger la BDD_Site2 à partir de SITE1.

Dans SITE1: SELECT * from BDD_SITE2.Table@LINKSITE2.

V-2 Transparence de localisation (SYNONYM)

Les synonymes fournissent à la fois l'indépendance des données et la transparence de l'emplacement. Synonymes permet principalement de cacher la localisation des tables distantes et donc de n'utiliser dans les requêtes que les noms des tables distantes.

```
CREATE [PUBLIC] SYNONYM BDD_SITE2.Table
FOR BDD_SITE2.Table@LINKSITE2.
```

SELECT * FROM BDD_SITE2

V-3 Manipulation des données dans une BDD Oracle répartie

Voir TP5